Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2007-03

Static reachability analysis and validation regarding security policies implemented via packet filters

Kantz, Stephen M.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/3586

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

STATIC REACHABILITY ANALYSIS AND VALIDATION REGARDING SECURITY POLICIES IMPLEMENTED VIA PACKET FILTERS

by

Stephen M. Kantz

March 2007

Thesis Advisor:	Geoffrey Xie
Second Reader:	Richard Riehle

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Static Reachability Analysis and Validation regarding Security Policies Implemented via Packet Filters		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Stephen M. Kantz			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** **N/A**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; Distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

The ability to statically determine what kinds of packets can be exchanged between two hosts on a network is desirable to those who design and operate networks, but this is a difficult and complex problem. Factors affecting reachability analysis are packet filters, routing policies and packet transformations. The number of variables within and among networks is intractable for manual computation. A proposed solution to this mess is a tractable framework for which to map networks into, thus creating a single unified model for analysis. It depends heavily on the use of transforming the problem into a classical graph problem that can be solved with polynomial time algorithms such as transitive closure.

This research develops an automated validation process to test the reachability upper bound calculated from a recent implementation of the framework which focuses specifically on the packet filter aspect, namely access control lists. Real-world network configuration files and network packet flow data from a Tier-1 Internet Service Provider is supplied as the data set. A significant contribution of this thesis is the application of real-world data to the proposed method for static reachability analysis as it pertains to the static testing of security policies applied via packet filters.

14. SUBJECT TERMS Static Reachability Analysis, Network Configuration, Access Control List, Packet Filter, Security Policy, Router Configuration, IP Networks, Unified Model, Framework			**15. NUMBER OF PAGES** 69
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

STATIC REACHABILITY ANALYSIS AND VALIDATION REGARDING SECURITY POLICIES IMPLEMENTED VIA PACKET FILTERS

Stephen M. Kantz
Lieutenant, United States Navy
Bachelor of Science (Biotechnology), Virginia Polytechnic Institute and State University, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2007

Author: Stephen M. Kantz

Approved by: Geoffrey Xie
Thesis Advisor

Richard Riehle
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The ability to statically determine what kinds of packets can be exchanged between two hosts on a network is desirable to those who design and operate networks, but this is a difficult and complex problem. Factors affecting reachability analysis are packet filters, routing policies and packet transformations. The number of variables within and among networks is intractable for manual computation. A proposed solution to this mess is a tractable framework for which to map networks into, thus creating a single unified model for analysis. It depends heavily on the use of transforming the problem into a classical graph problem that can be solved with polynomial time algorithms such as transitive closure.

This research develops an automated validation process to test the reachability upper bound calculated from a recent implementation of the framework which focuses specifically on the packet filter aspect, namely access control lists. Real-world network configuration files and network packet flow data from a Tier-1 Internet Service Provider is supplied as the data set. A significant contribution of this thesis is the application of real-world data to the proposed method for static reachability analysis as it pertains to the static testing of security policies applied via packet filters.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. OBJECTIVE 2
B. BENEFITS OF STATIC ANALYSIS 3
C. RESEARCH QUESTIONS 4
D. ORGANIZATION 4

II. BACKGROUND 5
A. STATIC REACHABILITY ANALYSIS FRAMEWORK MODEL 5
1. Formal Definition of Reachability Upper Bound 6
2. Reachability Upper Bound Calculation 6
B. METHODOLOGY AND DESIGN OF THE STATIC REACHABILITY ANALYSIS TOOL 7
1. The PacketSet Data Structure 7
2. Creation of a PacketSet 8
3. Reachability Upper Bound Computation 10
C. NETFLOW DATA SET 11

III. VALIDATION METHODOLOGY 13
A. REACHABILITY VALIDATION CORRECTNESS 13
B. AUTOMATED VALIDATION DESIGN 14
1. Main Algorithm 14
2. NetflowValidator Algorithm 15

IV. IMPLEMENTATION 17
A. SOFTWARE IMPLEMENTATION 17
1. Modifications to the Static Reachability Analysis Toolkit 17
a. GUI Replacement 17
b. Modifications in Parsing 18
c. Time and Process Efficiency Implementation 18
2. Automated Router Selection and Validation 19
B. VALIDATION RESULTS AND ANALYSIS 19
1. Validation Results 20
a. Memory and Space Complexity 20
b. Time Complexity 21
2. Results Analysis 21

V. RECOMMENDATIONS AND CONCLUSION 25
A. RECOMMENDATIONS FOR FUTURE WORK 25
B. CONCLUSION 26

APPENDIX A. NETWORKGRAPH2.JAVA 27

APPENDIX B. MAIN.JAVA 31

APPENDIX C. PORTDATABASE.JAVA 37

APPENDIX D. PROTOCOLDATABASE.JAVA 39

APPENDIX E. PATHCHOOSEREVOLVED.JAVA 41
APPENDIX F. SRADATABASE.JAVA..... 43
APPENDIX G. NETFLOWVALIDATOR.JAVA..... 45
LIST OF REFERENCES..... 51
INITIAL DISTRIBUTION LIST 53

LIST OF FIGURES

Figure 1. NetFlow Ingress and Egress 12

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Validation Error Cases 13
Table 2. Mathematical Description of RUB Validation Error Types 14
Table 3. Validation Results 20
Table 4. Processing Times 20

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my most sincere and heartfelt appreciation to all of my family and friends for their unwavering support, encouragement and motivation behind getting this thesis accomplished.

A special thanks to KnyteTyme for without whom this would not be possible.

Thanks to my advisor Geoff Xie for constantly challenging me to attain my academic best, his guidance and involving me in this unique research opportunity.

“Dômo arigatô gozaimasu” Sensei Riehle for helping me to maintain focus on what lies ahead.

Much thanks to my short-time associates at the Tier-1 ISP for their cooperation and insight which contributed to this research.

"Thank God -- every morning when you get up -- that you have something to do which must be done, whether you like it or not. Being forced to work, and forced to do your best, will breed in you a hundred virtues which the idle never know.” ~ Charles Kingsley

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The concept of a network dates back to primitive society where smoke signals, fire towers, and symbols viewed by spyglass were used to communicate and share information [1]. This basic idea of being able to reach others via an indirectly connected mechanism (be it physical or logical) is manifested by the complex computer networks commonly found today. They are now critical systems supporting a multitude of applications depended upon by businesses, governments, educational and private sectors. At their core is the premise of reachability between hosts.

Network reachability analysis is focused on identifying what types of packets are able to traverse between two hosts on a network beyond that of physical connectivity, network topology and routing protocols. It is subject to policies that are set in place for reasons such as security, network control, and licensing agreements, which in turn levy constraints on network design and restrict the number of destinations with which hosts are permitted to communicate. This poses a significant challenge for network designers who must properly fulfill security policy requirements.

Implementation of policy objectives involves three general methods – packet filters, routing policies and packet transformations – each has an affect on reachability and thus must be taken into consideration [2]. Determining reachability can thus become quite difficult. The problem is even further exacerbated by the size and complexity of the network itself.

Critical to the network's reliability is how faithfully the network design implements security policy. Identification of vulnerabilities in policy can be quite challenging. Given a scenario where a communication failure exists between two hosts, troubleshooting efforts may first seek to investigate the connectivity between them only to find that it remains intact. The next logical approach is to explore the network configuration for design or implementation flaws.

Attempted solutions to the problem are limited. Current practices of determining network reachability rely on real-time, intrusive, manual troubleshooting methods. Tools such as *ping* and *traceroute* send probe traffic over the network which in turn must be analyzed by hand through poring over the network configuration. Consider that campus, enterprise and backbone networks range from 5 to upwards of 1,000 routers, manual calculation is quite impractical [2]. Other tools analyze the flow of packets over a network (dynamic analysis) but require direct access to live networks working in real-time in order to monitor behavior [3]. They do not consider all of the *potential* packets that could traverse the network between hosts.

The benefits and practicality of an automated, static, non-intrusive method for reachability analysis have only recently been researched. This thesis focuses on the validation of a current implementation which analyzes packet filters in particular. It is based on the seminal research described in *On Static Reachability Analysis of IP Networks* conducted by Xie, et al., where a common framework is introduced to address three aspects of static reachability: packet filters, routing information and packet transformations.

A. OBJECTIVE

It is highly desirable to be able to statically determine what kinds of packets can be exchanged between two hosts on a network, however, this is difficult to do. The problem of reachability analysis must take into account more than routing protocols and topology. It must also consider packet filters, routing policies and packet transformations within that determination. Naturally, this is a difficult and complex problem for those who design and operate networks. The number of variables involved in a network and amongst networks can seem intractable and is simply too large for manual computation. An attempt at solving this mess is to develop a tractable framework for which to map networks into, thus creating a single unified model for analysis. It depends heavily on the use of transforming the problem into a classical graph problem that can be solved with polynomial time algorithms such as transitive closure [2].

This research attempts to validate the upper bound reachability results calculated from a recent implementation of the framework [4] which focuses on the packet filter aspect in particular, namely access control lists (ACLs). Real-world network configuration files as well as network packet flow data from a tier-1 Internet Service Provider (ISP) is supplied as the data set. A significant contribution of this thesis is the application of real-world data to the proposed method for static reachability analysis as it pertains to the static testing of security policies applied in part via packet filters (*i.e.,* ACLs).

B. BENEFITS OF STATIC ANALYSIS

Static analysis uses a white-box approach where at any given time a snapshot of the network's router configuration files is taken and then parsed into the framework thus serving as the source of information regarding the network under analysis. In this way, analysis can be conducted without direct interaction with the network itself and all of the critical reachability information regarding packet filters, routing policies and packet transformations is easily obtained.

Given that the network description can be gathered from the router configuration files, it is possible to test reachability independently. By turning the focus away from analysis on the live networks, which is done on a single IP basis, more thorough analysis can be conducted through identification of IP ranges via static analysis. Tools such as *ping* and *traceroute* are limited in that they test the specific probe traffic sent between only two hosts and according to the path designated by routing protocols. Rather, static analysis enables the determination of all potential packet sets that could traverse the network between two points as well as those routers and hosts a given packet set could reach regardless of what routing protocols dictate.

Validation of network design and security policies is another advantage of static reachability analysis. It can aide in meeting design goals of new networks prior to operational deployment or even before construction of the physical network. Additionally, static analysis provides the ability to conduct "what if"

scenarios on a network's reachability. This reduces the risk of disruption to live networks by predicting the impact of hardware failures, configuration changes, and maintenance schedules.

C. RESEARCH QUESTIONS

Is the recent automated static reachability analysis implementation [4], which focuses particularly on the packet filters (*e.g.* ACLs) aspect of the framework described in [2], valid when tested against known tier-1 ISP network packet flows?

a. Is it a viable suitable solution to validating security policies?

b. What improvements are necessary?

D. ORGANIZATION

In Chapter II, a background description of the current tool is provided to include its methodology. Chapter III delves into the validation method design and computation. It also explains the general characteristics of the ISP data sets used for validation and assumptions made. Chapter IV explains modifications made to the initial program source code as well as the development of new code. The limitations and bottlenecks discovered during the process are also noted as the validation results are reviewed. Lastly, Chapter V concludes the research and makes some recommendations for future work.

II. BACKGROUND

The framework described in [2] is a critical component of the logic implemented in the program developed to conduct static reachability analysis [4] upon which was the subject of validation and research for this thesis. As such, a brief introduction is necessary for the reader to better understand the concepts involved. Thereafter will be a description of the methodology and design of the program titled Static Reachability Analysis (SRA) Toolkit [4] used to compute the reachability upper bound for each router pair. The data set used for validation will conclude this chapter.

A. STATIC REACHABILITY ANALYSIS FRAMEWORK MODEL

The authors of [2] have developed a tractable framework for which to map networks into, thus creating a single unified model for analysis. It depends heavily on the use of transforming varying views of a network into a classical single graph problem that can be solved with polynomial time algorithms such as transitive closure [5], [6]. The graph accounts for packet filters (routers and links), routing policy (route redistribution) and packet transformations (virtual node-edge pair) as nodes where reachability is an edge metric. They formally define the graph G = (*V*, *E*, $\mathcal{F}_{\cdot}$) where *V* is the set of routers, *E* is the set of connecting edges, and $\mathcal{F}$ is an edge labeling function. For each edge $\langle u, v \rangle \in E$, $F_{u,v} \in \mathcal{F}$ represents the policies controlling the flow of packets between interfaces u and v or in reachability terms $F_{u,v}$ represents the set of packets that the network is able to carry between two nodes. In the case of this research, $f_{u,v}$ represents a packet filter containing predicates which challenge the properties of packet p to determine if it belongs in the set $F_{u,v}$ [2].

1. Formal Definition of Reachability Upper Bound

The most important computation for this research is the reachability upper bound (RUB) between a source and destination router, but unless they are directly connected there must be intermediate routers or multiple hops. Thus, the reachability for each intermediate hop represents a subset of packets that the network will allow between two routers i and j, denoted as $R_{i,j}$, over edge u and v since clearly not all packets traversing between them originated from a single source router. Since ACLs are applied per interface, the edge u and v represents the outbound and inbound interface respectively.

Routing protocols, link failures, and changes in routing advertisements have an affect on $R_{i,j}$. Each router has a mechanism called its Forwarding Information Base (FIB) by which it can inform others of changes and updates to the interfaces that should be used for sending packets. This is called a router's forwarding state, denoted by s. The set of all forwarding states for the entire network is denoted by S. The RUB, $R^U_{i,j}$, represents the set of packets that could *potentially* flow from i to j given that all possible changes in forwarding state were properly made by the FIBs . It is formally defined as:

$$R^U_{i,j} = \bigcup_{s \in S} R_{i,j}(s)$$

2. Reachability Upper Bound Calculation

Computation of the RUB applies the basic mathematical concepts such as transitive closure [5] and shortest path algorithms [6]. The following algorithm is used to calculate $R^U_{i,j}$ from source router i to destination router j that take up to m hops over the set of routers V. For each hop, intermediate calculations in reachability are represented by $R'(i, j)$ which consider the packet filter rules at each intermediary router k. Step 6 applies transitive closure of each $R'(i, j)$.

```
1. Initialize R(i,j) to F_{i,j} for all i;
2. for (m = 1 to ||V|| − 2) do
3.   for (i = 1 to ||V||) do
4.       R'(i,j) = ∅;
5.       for (k = 1 to ||V|| ) do
6.         if (< i,k >∈ E )
               then R'(i,j) = R'(i,j) ∪ {F_{i,k} ∩ R(k,j)};
7.   R(i,j) = R'(i,j);
```

B. METHODOLOGY AND DESIGN OF THE STATIC REACHABILITY ANALYSIS TOOL

The tool developed by [4] is an implementation of the static analysis concepts and algorithms previously discussed for computing the reachability upper bound of a network. This thesis research made a closer inspection of the tool as it performed against a real-world tier-1 ISP data set. Thus, an explanation of the methodology and design of the tool as to its handling of packet filters, data structures, and operations for reachability computation is in order.

1. The PacketSet Data Structure

The PacketSet data structure represents a packet filter (ACL) using a 5-tuple set notation. The value of each dimension is a range of the values (lower to upper) that are effectively permitted by each rule in the ACL. Additionally, unlike packet filters where each rule must be challenged sequentially in the order which they are listed, the PacketSet data structure eliminates this requirement. This is accomplished by mapping all permit and deny packet filter rules into the 5-tuple permit-only PacketSet data structure. Each 5-tuple is denoted using the following set notation:

[$\text{src-ip}_{\text{lower}}$, $\text{src-ip}_{\text{upper}}$];	(Source IP Address)
[$\text{src-port}_{\text{lower}}$, $\text{src-port}_{\text{upper}}$];	(Source Port Number)
[$\text{dest-ip}_{\text{lower}}$, $\text{dest-ip}_{\text{upper}}$];	(Destination IP Address)
[$\text{dest-port}_{\text{lower}}$, $\text{dest-port}_{\text{upper}}$];	(Destination Port Number)
[$\text{prot}_{\text{lower}}$, $\text{prot}_{\text{upper}}$]	(Protocol Number)

2. Creation of a PacketSet

After a router configuration file is parsed, each of the packet filter rules composing a single ACL are processed into one PacketSet data structure. The algorithm handles each rule sequentially. All permit rules are placed into a buffer until a deny rule is encountered, which is then placed into a separate buffer. Each subsequent permit rule is then checked against every preceding rule in the deny buffer. This invokes the GetPermitTuple function to map permit and deny rules into a new permit-only tuple. Each comparison will result in the creation of zero, one, or two new tuples based on the following scenarios:

0) The range of values in the permit rule is a subset of the deny rule's

1) Only one end (upper or lower range) of the permit and deny overlap

2) The range of values in the deny rule is a subset of the permit rule's

Note that given a d-dimension tuple, the most that can be created is 2d.

Creation of a PackSet data structure algorithm:

```
 1. Create empty PacketSet;
 2. Create an empty Deny Buffer;
 3. For (each rule in ACL), do
 4. Create an empty Interim Buffer;
 5. Convert rule into a new Current Tuple;
 6. If rule is "Deny",
    add Current Tuple to Deny Buffer;
 7. Else if rule is "Permit"
 8. Add Current Tuple to Interim Buffer;
 9. For (j=0 to size of Deny Buffer), do
10. For (k=0 to size of Interim Buffer), do
11. Perform a GetPermitTuple on Interim
    Buffer[k] against Deny Buffer[j];
12. Add Interim Buffer to PacketSet;
13. If PacketSet is not empty, perform
    optimization to merge and remove overlapping
    ranges;
```

In terms of mathematical notation, consider the following simple case involving only a two dimensional tuple (Although integers are used in the examples, for demonstrational purposes the first tuple could represent the source IP address range and the second tuple the source port number range):

For deny rule A and permit rule B:	Example
$[(A,1)_{lower},(A,1)_{upper}]; [(A,2)_{lower},(A,2)_{upper}]$	[39,62];[42,59]
$[(B,1)_{lower},(B,1)_{upper}]; [(B,2)_{lower},(B,2)_{upper}]$	[37,68];[25,81]

Each comparison seeks the nearest value outside of the bounds of the deny rule range per dimension:

$[(B,1)_{lower}, \min\{(A,1)_{lower} - 1, (B,1)_{upper}\}];$ $[(B,2)_{lower},(B,2)_{upper}] \cup$	[37,38]; [25,81] $\cup$
$[\max\{(B,1)_{lower},(A,1)_{upper} + 1\}, (B,1)_{upper}];$ $[(B,2)_{lower},(B,2)_{upper}] \cup$	[63,68]; [25,81] $\cup$
$[(B,1)_{lower},(B,1)_{upper}];$ $[(B,2)_{lower}, \min\{(A,2)_{lower} - 1, (B,2)_{upper}\}] \cup$	[37,68]; [25,41] $\cup$
$[(B,1)_{lower},(B,1)_{upper}];$ $[\max\{(B,2)_{lower},(A,2)_{upper} + 1\}, (B,2)_{upper}]$	[37,68]; [60,81]

The PacketSet data structure is extensible in that the emergence of a pattern appears given an increase in the number of dimensions. Overlapping tuple ranges can occur and thus an optimization function is necessary to compensate. The optimized pattern takes the following form:

```
[non-overlapping 1st-D permit range];
[2nd-D permit range]; … ; [xth-D permit range]

[overlapping 1st-D range];
[non-overlapping 2nd-D permit range];
[3rd-D permit range]; … ; [xth-D permit range]

[overlapping 1st-D range]; [overlapping 2nd-D range];
[non-overlapping 3rd-D permit range];
[4th-D permit range]; … ; [xth-D permit range]
```

Once the ACLs are successfully mapped into PacketSet data structures, the burden of sequential rule processing for each permit and deny rule is relieved. Set operations can then be easily performed to conduct static reachability analysis.

3. Reachability Upper Bound Computation

After the packet filters are properly accounted for by the PacketSet data structure, the reachability computation can then be conducted. However, since only router configuration files are used as the source of network information it is necessary to estimate network topology. This is done using a simple neighbor discovery based on the IP address and mask assigned to each interface. Those network prefixes that match between interfaces on varying routers are thus considered to be connected. The reachability algorithm can now traverse the network.

The tool uses an adaptation of the framework algorithm [2] to incorporate the PacketSet data structure. An algorithm is used to determine $F_{i,j}$, the set of packets permitted over an edge $\langle i, j \rangle$ based on the ACLs encountered on each interface $\langle u, v \rangle$ respectively.

```
1. For (each edge <i,j> in network) do
2. Initialize Fi,j to empty PacketSet;
3. For (each physical link <u,v>) do
4. Obtain all the ACLs activated by outbound
   queue of u;
5. PacketSet S1 = Intersection of all PacketSets
   specified by the ACLs obtained at step 4;
6. Obtain all the ACLs activated by inbound
   queue of v;
7. PacketSet S2 = Intersection of all PacketSets
   specified by the ACLs obtained at step 6;
```

8. $F_{i,j} = F_{i,j} \cup (S1 \cap S2)$

The next algorithm first determines the reachability from all sources given a single destination j, then it returns the RUB calculation based on the specified source router i as new PacketSet.

```
1. Initialize packetSetRUB[i][j] for all i:
     to Fi,j if i and j are neighbors;
     to whole PacketSet, if i=j;
     to empty PacketSet, otherwise;
2. for ( m=0 to (numberOfRouters - 3) ) do
3.   for ( i=0 to (numberOfRouters - 1) ) do
4.     tempPacketSetRUB[i][j] = empty;
5.     for (each interface (z) on router i) do
6.       for ( k=0 to numberOfRouters - 1 ) do
7.         if (k has an interface that is a
               neighbor of i on interface z)
8.             intersectedPacketSetRUB =
               Fi,k ∩ packetSetRUB[k][j];
9.             tempPacketSetRUB[i][j] =
               tempPacketSetRUB[i][j] ∪
               intersectedPacketSetRUB ;
10.        endif;
11.    endloop;
12.  packetSetRUB[i][j] = TempPacketSetRUB[i][j];
```

It is important to note that only packet filters are considered as the focus of research, thus the effect that a routing protocol might have on forwarding packets to neighbors is not. Rather, all permitted packet sets from a router are assumed to reach all of its neighbors.

C. NETFLOW DATA SET

The data set used for the validation of the computed reachability upper bound is referred to as NetFlow data. It is a Cisco Internet Operating System technology that collects information on actual packet flows traversing the network as it enters specified routers. Each router identifies a flow by unique characteristics, such as IP address and application, and caches the accumulated data until the flow is completed. [7]

Flow collection is not exhaustive however due to the resources that would be required for collection, processing and storage of the data. In order to control the amount of data collected, a technique called smart sampling preferentially samples larger flows such that an accurate representation is obtained from a subset of total flow while maintaining statistical variance. This is possible

because a large fraction of network traffic is contained within a small subset of flows. Smart sampling utilizes one of two methods. Threshold sampling [8], [9] is based on the characteristics of the packet stream and priority sampling [10], [11] which seeks a fixed number of the highest ranked samples found in a population.

A NetFlow data record is a simple text file where each line represents a single flow and each metric is comma delineated. It contains many metrics to which the full extent go beyond the scope of this research. Rather, only the pieces of information that fit the PacketSet 5-tuple data structure were extracted. The parsed NetFlow data record was maintained using the following line format:

```
{Source IP, Source Port Number,
Destination IP, Destination Port Number,
Protocol,
Ingress Router:Inbound Interface,
Egress Router:Outbound Interface}
```

The reason NetFlow data was selected to conduct the validation testing is because it represents packet flows that have successfully passed the security policies implemented on the network. It fits nicely into the previously described framework where the set of packets permitted from ingress router i to egress router j is based on the ACLs encountered on each interface $\langle u,v \rangle$ respectively (Figure 1).

Figure 1. NetFlow Ingress and Egress

III. VALIDATION METHODOLOGY

The validation methodology is a rather straightforward comparison between the reachability upper bound and the NetFlow data. Cases where a flow is found to fail validation requires some discussion as there could be various reasons and contributing factors. Also of importance are the specific validation results that this research most conclusively aims to discover. The design of the automated validation process is outlined as well.

A. REACHABILITY VALIDATION CORRECTNESS

The terms of reachability validation require some qualification as to the definition of correctness. There are four error cases that must be considered when interpreting the meaning of validity, but there two probable reasons (Table 1). The first reason can be due to a bug or discrepancy found within the RUB computation implementation. The second error type accounts for limitations that might be introduced as a result of not modeling routing protocols into the RUB computation, but does not leave out the possibility of a bug in the software either.

Error Type	Error Description	Possible Reason
False Deny 1	Flow mistakenly excluded from reachability upper-bound	Software bug since routing only decreases reachability
False Permit 1	Flow mistakenly included in reachability lower-bound	
False Deny 2	Flow mistakenly excluded from reachability lower-bound	Limitation due to not modeling routing protocols or due to software bug
False Permit 2	Flow mistakenly included in reachability upper-bound	

Table 1. Validation Error Cases

An early assumption is made which maintains the fidelity of NetFlow data such that the flows it represents encountered packet filter rules that were implemented correctly. Alternatively stated, all of the NetFlow data is expected to be found within the RUB provided the computation is correct. This is a fair

assumption which places trust in the tier-1 ISP network design while ensuring that the scope of the research remains on validation of the RUB computation. Based on that premise, only the False Deny 1 and False Permit 2 error cases apply. Though the latter case can feasibly occur, it cannot be conclusively determined within the scope of this research because the effects of routing protocols are not considered into the reachability computation. Therefore, the only conclusive result that can be determined from this research is in the False Deny 1 error case since the only possible reason stems from a bug in the RUB software.

A mathematical representation of the False Deny 1 and False Permit 2 error cases is in Table 2 where N represents the set of all potential packets permitted to traverse the network and f is the NetFlow data.

Error Type	Mathematical Description	Result
False Deny 1	$f \in N, f \notin RUB$	Bug in code
False Permit 2	$f \notin N, f \in RUB$	Inconclusive

Table 2. Mathematical Description of RUB Validation Error Types

B. AUTOMATED VALIDATION DESIGN

Since the RUB computation is represented by a range of accepted values, a simple check against each data field of the RUB computation is conducted to determine if the corresponding field of the NetFlow data is within those bounds. A flow is considered valid if all of its information falls within the bounds of the RUB. Otherwise it is considered to have failed validation.

1. Main Algorithm

A new Main function was developed in order to organize output, direct access to data sets for processing (e.g. NetFlow data and router configuration files) and begin the validation test. To increase efficiency, information such as

the parsed NetFlow data and the processed network configuration are saved to a file. Future runtime using the saved network configuration file is reduced from near 5 hours to only 20 minutes.

The parsed NetFlow data is output to a file which is then read into the program by the NetflowValidator function. (The details of which will be introduced in the following section.) A benefit of this allows for custom files to be created for the purposes of conducting tests on specific security policies, simulate network traffic ("what-if" tests) or to inject known packet flows that were not sampled by NetFlow. A brief algorithm explains the top level Main function:

```
1. Initialize data and results directories;
2. Check for parsed NetFlow file;
3.   if it exists continue;
4.   else parse raw NetFlow data and save file;
5. Check for saved router configuration;
6.   if it exists load file;
7.   else parse router configuration files;
8.       Conduct neighbor discovery;
9.       PacketSet creation;
10.      Save file;
11. Invoke NetflowValidator
```

2. NetflowValidator Algorithm

The validation process is handled by a new function called NetflowValidator. It is invoked after the raw NetFlow files are parsed and all network configuration PacketSets have been created. As each line of NetFlow is read, a RUB computation is conducted based on the ingress and egress routers identified per flow but only if it has not already been completed. This check is in place to increase efficiency since it is possible that the same pair of routers can be used for various flows. There is no need to compute the RUB again.

In some cases the actual egress router used by the flow was discernable by the NetFlow technology which resulted in multiple egress routers for a single

ingress router reported in the data record. This is handled by computing the RUB and validation test for each pair of routers.

Also included is a simple check to determine if a path exists (connectivity) between the ingress and egress routers based on the computed PacketSets. An open source Java based networking utility simulating Dijkstra's algorithm was used for this operation [12]. Initially implemented as an efficiency mechanism, it also serves to highlight possible errors in router configuration processing or raise connectivity concerns. (Appendix A)

The following algorithm of the NetflowValidator function provides more detail:

```
1. For each line of NetFlow data do;
2. Check for multiple egress routers;
3. if single egress router do
4.   RUB check;
5.     if previously computed, load, continue;
6.     else check for path, compute RUB and save;
7. if multiple egress routers do;
8.   RUB check for each egress router;
9.     if previously computed, load, continue;
10.    else check for path, compute RUB and save;
11. Conduct validation test;
12.    for each data field;
          {Source IP Address Range;
          Source Port Number Range;
          Destination IP Address Range;
          Destination Port Number Range;
          Protocol Number Range}
13.       if NetFlow data field exceeds bounds of
          PacketSet data field return false;
14.       else return true;
15. Output and save results;
```

IV. IMPLEMENTATION

The development and actual implementation of the validation research is explored in this chapter. Changes to reachability software, development of new code, utilization of existing code as well as limitations and bottlenecks discovered are outlined. Then the validation results are introduced and analyzed. Lastly, a discussion is had on recommended future work and conclusions.

A. SOFTWARE IMPLEMENTATION

Since the data sets for this research were provided by a Tier-1 ISP, the proprietary information was required to remain on their systems. All of the software involved in this research was executed on the ISP research systems via a secure remote connection. The system was a sparc SUNW, Sun-Fire-15000 processor platform running SunOS 5.8 that had approximately 204 Gigabytes of memory and 54 Central Processing Units. The software was implemented using Eclipse SDK version 3.2.2 and compiled using Java version 1.5.0_06-b05. Perl version 5.005_03 was used in executing the NetFlow parser script which was provided by the ISP technicians.

1. Modifications to the Static Reachability Analysis Toolkit

Changes to the original software [4] were necessary in order to create an automated validation process using the NetFlow data within the confines of the collaborating ISP remote system. Further alterations were required to improve efficiency as a result of the complexity introduced by such a large data set. Few modifications were made to the parsing process in order to facilitate the process. No changes were made to the implemented logic for computing the reachability upper bound.

a. GUI Replacement

The Toolkit code utilized a Graphical User Interface (GUI) for the input of the router configuration data set and the selection of source and destination routers for the reachability analysis. This was deemed impractical for

the purposes of this research considering that an automated solution was desired for such a large data set. In place of the GUI, the Main function described in Chapter III was created (code is provided in Appendix B) to handle program startup, organization of input and output files, and invoke the parsing and automated validation processes.

b. Modifications in Parsing

During trials with the ISP router configuration files, parsing errors were encountered with the proper translation of protocol and port names to their corresponding numerical value designated by the Internet Assigned Numbers Authority. The reason was found to be related to variances used by Cisco in the shorthand notations of the protocol and port names. Another factor was that the list of number conversions included in the code was quite limited.

The solution was to create two functions which served as a port and protocol name to number conversion database. It is intended to be a near exhaustive list extracted from those found within system files (see Appendix C for the PortDatabase function). Due to further variances however, a custom file had to be constructed for the protocol numbers. As such, it is required to be located in the same directory from which the validation program is executed (see Appendix D for the ProtocolDatabase function).

c. Time and Process Efficiency Implementation

Two features of Java were implemented to take advantage of the computation power and large memory of the computer system used for this research. The total time for PacketSet creation was significantly reduced by using a 300 thread pool. Additionally, serialization was implemented to streamline the processing of the network configuration files into PacketSet and Router Configuration objects. Router configuration file parsing as well as neighbor discovery also experienced faster total process times each with a 100 thread pool.

2. Automated Router Selection and Validation

Router selection automation is achieved by modifying the original PathChooser function to accept the ingress and egress routers listed per line of NetFlow data instead of those manually specified by a user via the GUI. The new function, named PathChooserEvolved (Appendix E), is invoked by a new function called SRADatabase (Appendix F) which tracks if a RUB computation between the two routers already exists. If it does not exist, then SRADatabase passes the ingress and egress routers obtained via the NetflowValidator function (Appendix G) to PathChooserEvolved for reachability computation. The following algorithm outlines this process:

```
1. Main function invokes NetflowValidator;
2. For each line of NetFlow data the
   NetflowValidator obtains the ingress and
   egress router names;
3. Router names are passed to SRADatabase to
   check if RUB computation exists;
4. if RUB computation exists;
5.    Load information and return to
      NetflowValidator for test;
6. else pass router names to PathChooserEvolved;
7.    PathChooserEvolved invokes RUB computation
      and returns information to SRADatabase for
      storage;
8.    Information retrieved by NetflowValidator
      for test;
9. Continue;
```

B. VALIDATION RESULTS AND ANALYSIS

The results of the automated validation program are rather straightforward. As previously mentioned, each line of NetFlow data represents a single flow that was tested against the static reachability upper bound calculation. A line is considered to have passed validation if all of its data field values fit within the range determined by the RUB. Those lines that were

determined to fail validation are analyzed for likely reasons. Limitations and bottlenecks that affected the validation process are also addressed therein.

1. Validation Results

Validation processed 100,826 lines of NetFlow data and computed 1,281 static reachability upper bound computations over a network of near 800 routers in about 10 days, 5 hours and 43 minutes. Final results revealed that approximately 66 percent of the NetFlow data passed validation where 34 percent failed. Tables 3 and 4 provide further details. Note that validation time is dominated by the time to compute each RUB (the time required for the actual validation check is negligible) and is separate from the time it takes to process the configuration files into the framework (PacketSets).

Result	Number of Lines	Percentage of Total Lines
Passed Validation	66,663	66%
Failed Validation	34,163	34%

Table 3. Validation Results

Process	Run Time
Network Configuration	4 hours, 50 min
Validation	10 days, 5 hours, 43 min
RUB Computation	11 min on average

Table 4. Processing Times

a. Memory and Space Complexity

The software is quite memory intensive. Initial trials of this research encountered OutOfMemory errors caused by reaching the default heap space that Java 1.5 allows. The network configuration must be maintained within memory as well as each RUB computation. Considering the magnitude of information that must be handled for a Tier-1 ISP, compounded at times by the

use of at most 300 concurrent processes, the likelihood of such an error was highly probable though originally unanticipated.

The system testbed was more than capable of providing an adequate amount of memory with over 200GB available. Through troubleshooting, 3GB was determined to be the most allowed by Java. Implementing the `-Xmx3g` option alleviated the problem.

b. Time Complexity

PacketSet creation is processing intensive since the time to process one ACL is proportional to the square of the total number of rules comprised within it. There were approximately 688 *discrete* ACLs processed for a combined total of 32,160 rules on this Tier-1 network of near 800 routers. Though the time elapsed is attributed to the processing of 70,297 *individual* ACLs for a combined total of approximately 3,313,607 rules.

The process of parsing the configuration text files and creating PacketSets into a representative network configuration averaged about 5 hours. If changes to the router configuration files are uncommon, this can be considered a one time cost of processing overhead because it is saved to an output file as implemented in the Main function. Subsequent runs required only 20 minutes to load the router configuration file.

A major bottleneck of the validation process is the RUB computation as each averaged about 11 minutes each. For a network size such as the Tier-1, it would take about two weeks to determine the reachability between all router pairs implementing efficiency methods such as a 300 thread pool. This was considered to be a worst case scenario however in that the NetFlow data only represents a subset of total flow. Hence, only the necessary reachability tests were conducted per NetFlow data.

2. Results Analysis

The two-thirds majority of valid NetFlow data is rather encouraging because it demonstrates that the static reachability analysis framework and

implementation functions as hoped. It is the remaining third of data that requires closer inspection as to the cause for validation failure. Based on previous discussion regarding error cases, the reason was expected to be found within the software.

A quick inspection of the failed NetFlow lines revealed no discrepancy that indicated a parsing error. Thus the focus of attention turned to an exploration of the ACLs that those packet flows would encounter on the network. This required a review of the RUB computation and the original router configuration files.

It was discovered that in each failed case randomly selected for inspection that no RUB was found between the router pair. For there to be no RUB computation would imply that none of the ACLs on both the ingress or egress router intersected, thus concluding no reachability rendering the line of NetFlow data to be erroneous. Surely this was not true based on the known characteristics of the NetFlow data.

The next logical step was to confirm the presence of ACLs applied to the router pair or possibly the lack thereof. However, the latter case where no filters are applied to a router interface is handled within the SRA software by assigning a `permit any` filter. If that were to have occurred, then the NetFlow line would have passed validation.

Both the PacketSet data and the router configuration file confirmed the presence of ACLs applied to each router pair. Close study of the syntax used in the router configuration files disclosed a critical discrepancy with the software. The version of Cisco IOS used by the Tier-1 ISP included the feature of IP named ACLs in addition to the traditional use of numbered designation [13]. The SRA software focuses only on the latter, which indicated that those IP named ACLs were not being properly accounted for in the RUB computation.

Since there were ACLs applied to the interfaces of ingress and egress routers, an implied `deny all` resulted for the remaining set of packets that

were not listed. Hence, because of this it can be said that those lines of failed NetFlow data depended on the missing packet filters in order to be recognized as valid.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RECOMMENDATIONS AND CONCLUSION

The final chapter of this thesis identifies areas for improvement of the current reachability software to include future work regarding the next stage of implementing the SRA framework. Final comments mark the conclusion of this research.

A. RECOMMENDATIONS FOR FUTURE WORK

Initial suspicion directed by the False Deny 1 error case proved to be correct. The existence of a bug in the software was indeed found. The SRA Toolkit only handles numbered ACLs. The fact that a third of the NetFlow data failed in relation to this illustrates the likelihood of encountering the usage of IP named ACLs in the future. Thus, it should be incorporated into the program. As it currently stands, implementation is designed to only handle integers. Modifications are required which will enable these ACLs to be properly parsed and processed into PacketSet data structures that handle strings as well.

The router configuration file parser could be made more efficient. Those interfaces that have a private or loopback IP address should be recognized and ignored. A considerable amount of interfaces were actually named `Loopback`. The ability to identify these for exclusion is recommended as well. If insignificant items such as these are not considered for processing, time and memory complexity would be reduced to a degree.

Performance could potentially be improved by implementing a mechanism which saves the RUB computations in a similar design as was the router configuration in this research. This would enable future validation tests to be conducted more quickly assuming no changes were made to the network. Once routing protocol or policy is implemented into the program, it may be possible to only update the RUB database even if changes are made.

B. CONCLUSION

This research served an important role in the development of an automated solution for static analysis of network reachability as it performed the first rigorous test of the only known SRA tool implementation [4]. With the majority of NetFlow lines found valid, static reachability analysis has been demonstrated to be a viable solution suitable for security policy validation. The creation of an automated validation mechanism provides the basis for testing future work as a complete solution is sought.

Completion of this stage of development is highly anticipated. The next step in this research should be to implement the reachability lower bound with regards to packet filters. Once this is accomplished, along with previously mentioned modifications, yet another automated validation test will be required. Pending those results, the packet filter aspect of network reachability will have been properly addressed.

The benefits of incorporating the affect of routing policies on network reachability are apparent even in this research. It is the next logical aspect of static reachability analysis that can be modeled into the proposed framework. A rather powerful tool will result when combined with the packet filter component. Though Java was the preferred tool of implementation in this research, more efficient and computationally powerful alternatives may prove to be beneficial in experimental evaluation regarding this aspect.

Research in static reachability analysis has proven successful thus far. Significant progress has been made through the development of a framework, creation of a working implementation regarding the packet filter aspect, and now the rigorous validation of such. The successful incorporation of routing policy and packet transformations to static analysis will result in a first of its kind solution to designing networks more efficiently while ensuring correct implementation though semantic verification.

APPENDIX A. NETWORKGRAPH2.JAVA

```
package StaticReachabilityAnalysis;

import StaticReachabilityAnalysis.*;

import java.awt.Font;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Random;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

import mascoptLib.abstractGraph.AbstractPath;
import mascoptLib.algos.abstractalgos.Dijkstra;
import mascoptLib.gui.*;
import mascoptLib.graphs.*;
import mascoptLib.io.graph.MGLReader;
import mascoptLib.io.graph.MGLWriter;

public class NetworkGraph2 {

      private HashMap nameToVertexMap = new HashMap();
      private DiGraph graph = null;

      public NetworkGraph2(NetworkConfig conf) {

            HashMap ipToPairMap = initMap(conf);
            VertexSet ns = new VertexSet();
            ArcSet as1 = new ArcSet(ns);


            Iterator routers = conf.tableOfRouters.values().iterator();

            while(routers.hasNext()) {
                  RouterConfig currentRouter =
                        (RouterConfig)routers.next();
                  Vertex v = new Vertex();

      nameToVertexMap.put(currentRouter.hostName.toLowerCase(), v);
                  ns.add(v);
            }

            routers = conf.tableOfRouters.values().iterator();

            while(routers.hasNext()) {
                  RouterConfig currentRouter =
                        (RouterConfig)routers.next();
```

```
                Iterator interfaces =
            currentRouter.tableOfInterfaceByNames.values().iterator();
                while(interfaces.hasNext()) {
                        InterfaceConfig iconf =
                                (InterfaceConfig)interfaces.next();
                        for(int i=0; i<iconf.neighbors.size(); i++) {
                                String ip =
                                        (String)iconf.neighbors.get(i);
                                InterfacePair ipair =
                                        (InterfacePair)ipToPairMap.get(ip);

                                Vertex v1 =
                (Vertex)nameToVertexMap.get(currentRouter.hostName);
                                Vertex v2 =
                (Vertex)nameToVertexMap.get(ipair.rconfig.hostName);

                                as1.add(new Arc(v1, v2));
                        }
                }
            }

            //create logical graph
        graph = new DiGraph(ns,as1);
        graph.setName(conf.networkName);
    }

    private HashMap initMap(NetworkConfig nconfig) {

            HashMap ipToPairMap = new HashMap();

            Iterator routers =
                nconfig.tableOfRouters.values().iterator();

            while(routers.hasNext()) {
                RouterConfig currentRouter =
                        (RouterConfig)routers.next();
                Iterator ips =
            currentRouter.tableOfInterfaceByIPs.keySet().iterator();
                while(ips.hasNext()) {
                        String ip = (String)ips.next();
                        InterfaceConfig iconfig =
(InterfaceConfig)currentRouter.tableOfInterfaceByIPs.get(ip);
                        if(ipToPairMap.containsKey(ip)) {
                                System.out.println(
                                "ERROR: NOT UNIQUE IPs ("
                                + currentRouter.hostName + ", "
                                + iconfig.interfaceName + ", "
                                + ip + ")");
                                continue;
                        }
                        InterfacePair pair = new
                                InterfacePair(currentRouter, iconfig);
                        ipToPairMap.put(ip, pair);
                }
            }

            return ipToPairMap;
```

```
	}

	public boolean doesPathExist(String src, String dst) {
		Dijkstra dj = new Dijkstra(graph);

		Vertex source =
(Vertex)nameToVertexMap.get(src.toLowerCase());
		Vertex destination =
(Vertex)nameToVertexMap.get(dst.toLowerCase());

		if(source == null) {
			System.out.println
("Source router " + src + " not found in NetworkGraph.");
			return false;
		}

		if(destination == null) {
			System.out.println
("Destination router " + dst + " not found in NetworkGraph.");
			return false;
		}

		if(source.getDegree(graph) == 0 ||
destination.getDegree(graph) == 0) {
			return false;
		}

		dj.valuateFromSource(source);

		AbstractPath p = null;
		try{
			p = dj.getShortestPathTo(destination);
		} catch(Exception e) {
			System.out.println
("CANT CALC PATH: " + e.getMessage());
		}

		return (p != null);
	}

	private static class InterfacePair {
		public InterfaceConfig iconfig;
		public RouterConfig rconfig;


		public InterfacePair(RouterConfig router, InterfaceConfig
iface) {
			iconfig = iface;
			rconfig = router;
		}
	}
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. MAIN.JAVA

```
package StaticReachabilityAnalysis;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Iterator;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;


/*********************************************************************
 * New main for non-GUI interface

*********************************************************************/

public class Main {

      private static final String CONFIGS_INPUT_BASE_DIR =
            "/ipso/netdb_1/cbb/";

      private static final String NETFLOW_COMMAND =
            "perl /ipso/project_6/reachability/netflow.pl";

      private static String parentDir;
      private static String SRAOutputDir;
      private static String parsedConfigsDir;
      private static String netflowOutputFile;
      private static String configsInputDir;

      private static String savedNetworkConfig;

      private static boolean DEBUG = false;

      private static void initDirectories(String year, String month,
      String day) {
            parentDir = "/ipso/project_6/reachability/" + year + "_" +
                        month + "_" + day + "-results/";
            SRAOutputDir = parentDir + "SRAOutput";
            parsedConfigsDir = parentDir + "parsedConfigs";

            File SRA = new File(SRAOutputDir);
            File parsedConfigs = new File(parsedConfigsDir);

            // create static reachability analysis directory
            if(SRA.exists() == false) {
                  boolean result = SRA.mkdirs();
```

```
            if(result == false) {
                  System.err.println("SRA dir creation failed!");
                  System.exit(-1);
                  }
            }

      //create parsedconfigs directory
      if(parsedConfigs.exists() == false) {
          boolean result = parsedConfigs.mkdirs();
          if(result == false) {
            System.err.println
                  ("parsedConfigs dir creation failed!");
                  System.exit(-1);
                }
            }

      netflowOutputFile = parentDir + "netflowOutput.txt";
      configsInputDir = CONFIGS_INPUT_BASE_DIR + year + month +
                        day + "/";
      savedNetworkConfig = parentDir + "networkConfig.sav.gz";

      System.out.println("SRA output dir: " + SRAOutputDir);
      System.out.println
      ("parsedConfigs output dir: " + parsedConfigsDir);
      System.out.println
      ("netflow.pl output file: " + netflowOutputFile);
      System.out.println
      ("configs input dir: " + configsInputDir);
      System.out.println
      ("NetworkConfig savefile: " + savedNetworkConfig);
}

private static void saveNetworkConfig(NetworkConfig config) {

      try {
            FileOutputStream fos = new
            FileOutputStream(savedNetworkConfig);
            BufferedOutputStream bos = new
            BufferedOutputStream(fos);
            GZIPOutputStream gzos = new GZIPOutputStream(bos);
            ObjectOutputStream oos = new
            ObjectOutputStream(gzos);
            oos.writeObject(config);
            oos.flush();
            oos.close();
      } catch (FileNotFoundException e) {
            e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }
}

private static NetworkConfig loadSavedNetworkConfig( ) {
      NetworkConfig result = null;

      try {
```

```
                FileInputStream fis = new
                FileInputStream(savedNetworkConfig);
                BufferedInputStream bis = new
                BufferedInputStream(fis);
                GZIPInputStream gzis = new GZIPInputStream(bis);
                ObjectInputStream ois = new ObjectInputStream(gzis);
                result = (NetworkConfig)ois.readObject();
                ois.close();
                return result;
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }

            return null;
        }

    private static void printTime(long millis) {
        long ms = millis % 1000;
        millis /= 1000;

        long seconds = millis % 60;
        millis /= 60;

        long minutes = millis % 60;
        millis /= 60;

        long hours = millis % 24;

        System.out.print(hours + "h " + minutes + "m " + seconds + "s "
                         + ms + "ms");
         }

//     java Main 2007 01 11
//     arg[0] = year
//     arg[1] = month
//     arg[2] = day

public static void main(String args[]) {
      String year = args[0];
      String month = args[1];
      String day = args[2];

      if(args.length < 3 || args.length > 4) {
            System.out.println("Syntax Error! Enter Month Day Year");
                  System.exit(-1);
            }
            if(args.length == 4) {
                DEBUG = true;
            }

            initDirectories(year, month, day);

            long wholeBefore = System.currentTimeMillis();
```

```
            try {

                  String cmd = NETFLOW_COMMAND
                  + " " + year
                  + " " + month
                  + " " + day
                  + " " + netflowOutputFile;

                  File netflowFile = new File(netflowOutputFile);

                  if(netflowFile.exists() == false) {
                  System.out.println("Running command: '" + cmd + "'");
                      Process child = Runtime.getRuntime().exec(cmd);
                      InputStream in = child.getInputStream();
                      int c;
                      while ((c = in.read()) != -1) {
                        System.out.print((char)c);
                        System.out.flush();
                      }
                  } else {
                  System.out.println
                  (netflowOutputFile + " already exists!");
                  }


                  File configsInput = new File(configsInputDir);
                  File parsedConfigsOutput =
                  new File(parsedConfigsDir);

                  NetworkConfig network = null;
                  File savedFile = new File(savedNetworkConfig);

                  if(savedFile.exists()) {
                        long before = System.currentTimeMillis();
                  System.out.println
                  (savedNetworkConfig + " exists - loading data!!");
                        network = loadSavedNetworkConfig();
                        long after = System.currentTimeMillis();
                  System.out.println
                  ("Loading file took " + (after-before) + "ms");
                  } else {
                        System.out.println
                        (savedNetworkConfig + " doesnt exist - parsing
                        config files!");
                        network = new NetworkConfig();
                        new Parser(network, configsInput,
                        parsedConfigsOutput);
                        saveNetworkConfig(network);
                        System.out.println("NetworkConfig saved");
                        //Enable data dump for debugging
                        new NetworkDataDump(network,
                        parsedConfigsOutput);
                  }

                  Iterator i =
network.tableOfRouters.keySet().iterator();
```

```
                    NetflowValidator nfv = new
NetflowValidator(netflowOutputFile, SRAOutputDir, network);

                    long validBefore = System.currentTimeMillis();
                    nfv.startValidation();
                    long validAfter = System.currentTimeMillis();

                    System.out.print("Validation took ");
                    printTime(validAfter-validBefore);
                    System.out.println();

              } catch (IOException e) {
                    e.printStackTrace(System.err);
              }
              long wholeAfter = System.currentTimeMillis();

              System.out.print("Whole process took ");
              printTime(wholeAfter-wholeBefore);
              System.out.println();
        }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. PORTDATABASE.JAVA

```
package StaticReachabilityAnalysis;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.DatabaseMetaData;
import java.util.HashMap;
import java.util.StringTokenizer;

public class PortDatabase {

      private static HashMap database = null;

      private static final String SERVICES_FILENAME
      = "/etc/services"; // aka /etc/inet/services
      // c:\winnt\system32\drivers\etc\services on Win NT/2000

      public static String getPortByName(String name) {
            if(database == null) {
                  database = new HashMap();
                  generateDatabase();
            }
            return (String)database.get(name);
      }


      private static void parseServicesLine(String line) {
            if(line.length() == 0 || line.charAt(0) == '#') {
                  return;
            }

            // Parse line
            StringTokenizer st = new
                  StringTokenizer(line, " \t/#");

            // First get the name on the line (parameter 1):
            if (! st.hasMoreTokens()) {
                  return; // error
            }
            String name = st.nextToken().trim();

            // Next get the service name on the line (parameter 2):
            if (! st.hasMoreTokens()) {
                  return; // error
            }
            String portValue = st.nextToken().trim();

            // Finally get the class on the line (parameter 3):
            if (! st.hasMoreTokens()) {
                  return; // error
            }
            String classValue = st.nextToken().trim();
```

```
        // else if(port.equalsIgnoreCase(".,..")) portNumber= "..";

        if(database.containsKey(name)) {
            return;
        }
        database.put(name, portValue);

    }     // parseServicesLine()



    private static void generateDatabase( ) {
        int port = -1;
        try {
            String line;
            BufferedReader br = new BufferedReader(
                                new InputStreamReader(
                                   new FileInputStream(
                                      SERVICES_FILENAME)));

            // Read /etc/services file.
            // Skip comments and empty lines.
            while ((line = br.readLine()) != null) {
                parseServicesLine(line);
            }     // while
            br.close();


        } catch (IOException ioe) {
            // File doesn't exist or is otherwise not available.
            // Keep defaults
            ioe.printStackTrace();
        }
    }
}
```

APPENDIX D. PROTOCOLDATABASE.JAVA

```
package StaticReachabilityAnalysis;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.DatabaseMetaData;
import java.util.HashMap;
import java.util.StringTokenizer;

public class ProtocolDatabase {

	private static HashMap database = null;

	private static final String SERVICES_FILENAME
	= "protocols"; // aka /etc/inet/services
	// c:\winnt\system32\drivers\etc\services on Win NT/2000

	public static String getProtocolByName(String name) {
		if(database == null) {
			database = new HashMap();
			generateDatabase();
		}
		return (String)database.get(name);
	}


	private static void parseServicesLine(String line) {
		if(line.length() == 0 || line.charAt(0) == '#') {
			return;
		}

		// Parse line
		StringTokenizer st = new
			StringTokenizer(line, " \t/#");

		// First get the name on the line (parameter 1):
		if (! st.hasMoreTokens()) {
			return; // error
		}
		String name = st.nextToken().trim();

		// Next get the service name on the line (parameter 2):
		if (! st.hasMoreTokens()) {
			return; // error
		}
		String portValue = st.nextToken().trim();

		// Finally get the class on the line (parameter 3):
		if (! st.hasMoreTokens()) {
			return; // error
		}
		String classValue = st.nextToken().trim();
```

```
        if(database.containsKey(name)) {
                return;
        }

        database.put(name, portValue);

    }     // parseServicesLine()




    private synchronized static void generateDatabase( ) {
        int port = -1;
        try {
                String line;
                BufferedReader br = new BufferedReader(
                                    new InputStreamReader(
                                       new FileInputStream(
                                          SERVICES_FILENAME)));

                // Read /etc/services file.
                // Skip comments and empty lines.
                while ((line = br.readLine()) != null) {
                        parseServicesLine(line);
                }     // while
                br.close();


        } catch (IOException ioe) {
                // File doesn't exist or is otherwise not available.
                // Keep defaults
                ioe.printStackTrace();
        }
    }

    public static void main(String args[]) {
        System.out.println(getProtocolByName("pim"));
    }
}
```

APPENDIX E. PATHCHOOSEREVOLVED.JAVA

```
package StaticReachabilityAnalysis;
import java.util.*;
import java.io.*;
import javax.swing.*;

public class PathChooserEvolved {
    NetworkConfig theNetwork = null;
    String theOutputDir = null;

    private int getRouterIndex(String routerName) {
      System.out.println("Trying to find router: " + routerName);

      Iterator routerList =
theNetwork.tableOfRouters.keySet().iterator();
      int i=0;
      while(routerList.hasNext()) {
            String router = (String)routerList.next();

            if(router.equalsIgnoreCase(routerName)) {
                  return i;
            }
            i++;
      }
      return -1;
    }

    /** Creates new form PathChooser */
    public PathChooserEvolved(NetworkConfig network, String outputDir)
{
        theNetwork = network;
        theOutputDir = outputDir;
    }


/*********************************************************************
*********
     *
     * Run the reachability computation code
     *

*********************************************************************
*******/
    public PacketSet ComputeNow(String ingress, String egress){
        boolean successComputeNow=false;
        boolean successFileSave=false;

        String sourceName = ingress;
        String destinationName = egress;

        PacketSet rUB = new PacketSet();
        PacketSet rLB = new PacketSet();
        /*************************
```

```
         * Display reachability calculation information on default
system.out display
         **************************/
        System.out.println("=== Calculating Reachability ===");

        // workaround for current code.
        int source = getRouterIndex(sourceName);
        int destination = getRouterIndex(destinationName);

      if(source == -1 || destination == -1) {
            System.out.println("DEBUG - Either source or destination
routers not in router index!");
          return null;
      }

        rUB = rUB.InitializePath(theNetwork,source, destination, rLB);

        /***************************
         * Save reachability calculation results to a file in the
output directory
         ***************************/
        String outputFileName = "SRA_" + sourceName + "_to_" +
destinationName + ".txt";

        System.out.println("Writing results to " + outputFileName);

        try {
            File outputFile = new File (theOutputDir, outputFileName);
            PrintWriter outFile = new PrintWriter (outputFile);
            outFile.println("Reachability Upper Bound from " +
sourceName + " to "
                    + destinationName + " : " + "\r\n");
            outFile.println(rUB);
            outFile.close();
            successFileSave = true;
        } catch (Exception e) { System.out.println ("Error - " + e); }

        successComputeNow = true;
        return rUB;
    }

    }

}
```

APPENDIX F. SRADATABASE.JAVA

```
package StaticReachabilityAnalysis;

import java.util.HashMap;
import java.util.LinkedList;

public class SRADatabase {

	// a map from a ingress/egress router pair to a packetset
	// that is generated from SRA analysis
	private static HashMap<String, PacketSet> database =
		new HashMap<String, PacketSet>();

	// NetworkConfig is generated from Parser
	private NetworkConfig networkConfig;
	// Where to put SRA files, even though we dont technically need
them
	private String SRAOutputDir;

	private LinkedList<String> noPathExistsList = new
LinkedList<String>();

	private NetworkGraph2 networkGraph = null;

	public SRADatabase(NetworkConfig config, String SRAOutputDir) {
		networkConfig = config;
		this.SRAOutputDir = SRAOutputDir;
		networkGraph = new NetworkGraph2(config);
	}

	// Returns a packetset if we have already generated it, otherwise
	// it is generated and placed in the map.
	public PacketSet getPacketSet(String ingressRouter, String
egressRouter)
	throws PathNotPossibleException {
		String key = ingressRouter + "-" + egressRouter;

		if(database.containsKey(key)) {
			return database.get(key);
		}

		if(noPathExistsList.contains(key)) {
			throw new PathNotPossibleException(ingressRouter,
egressRouter);
		}

		if(networkGraph.doesPathExist(ingressRouter, egressRouter)
== false) {
			noPathExistsList.add(key);

			throw new PathNotPossibleException(ingressRouter,
egressRouter);
		}
```

```
            PathChooserEvolved pce = new
PathChooserEvolved(networkConfig, SRAOutputDir);
            PacketSet ps = pce.ComputeNow(ingressRouter, egressRouter);

            if(ps == null) {
                return null;
            }

            database.put(key, ps);

            return ps;
      }

}
```

APPENDIX G. NETFLOWVALIDATOR.JAVA

```
package StaticReachabilityAnalysis;

//package StaticReachabilityAnalysis;

import java.io.*;
import java.text.*;
import java.util.ArrayList;

/*********************************************************************
* Reader function for parsed NetFlow data

*********************************************************************/

public class NetflowValidator {

      private File netflowFile = null;

      private SRADatabase sraDatabase = null;

      // MessageFormat for a NORMAL line in the parsed netflow file
      private static final MessageFormat mfNorm = new MessageFormat(
                  "{0},{1},{2},{3},{4},{5}:{6},{7}:{8}");

      // MessageFormat for an ABNORMAL (multiple egress) line in the
      parsed netflow file //
      private static final MessageFormat mfManyEgress = new
MessageFormat(
                  "{0},{1},{2},{3},{4},{5}:{6},{7}");

      NetflowValidator(String parsedNetflowFilename, String SRADir,
                  NetworkConfig network) throws IOException {
            netflowFile = new File(parsedNetflowFilename);
            sraDatabase = new SRADatabase(network, SRADir);
      }

      // returns a list of the egress routers
      private String[] parseMultipleEgress(String multEgress) {
            ArrayList<String> egressRouters = new ArrayList<String>();

            String[] parts = multEgress.split("%");
            for (int i = 0; i < parts.length; i++) {
                  String[] smallerParts = parts[i].split(":");
                  egressRouters.add(smallerParts[0]);
            }
            String[] result = new String[egressRouters.size()];
            result = egressRouters.toArray(result);
            return result;
      }

      // If return value has length 9 => normal
      // else abnormal (multiple egresses)
      // on error return null
      private Object[] parseLine(String line) {
```

```
            try {
                  if (line.indexOf("UNKNOWN") != -1) {
                        return null;
                  } else if (line.indexOf('%') != -1) {
                        Object[] parts = mfManyEgress.parse(line);
                        return parts;
                  } else {
                        Object[] parts = mfNorm.parse(line);
                        return parts;
                  }
            } catch (ParseException e) {
                  System.out.println
                  ("There was an error parsing the line: " + line);
                  e.printStackTrace();
            }
            return null;
      }

      // validate for multiple egresses
      private boolean validate(String srcip, String srcport, String
      dstip, String dstport, String protocol, String ingress, String[]
      egresses) throws RouterConfigNotPresentException,
      PathNotPossibleException {

            for (int i = 0; i < egresses.length; i++) {
                  if (validate(srcip, srcport, dstip, dstport,
                  protocol, ingress, egresses[i]) == true) {
                        return true;
                  }
            }

            return false;
      }

      // validate a single instance
      private boolean validate(String srcip, String srcport, String
dstip, String dstport, String protocol, String ingress, String egress)
throws RouterConfigNotPresentException, PathNotPossibleException {

            PacketSet ps = null;

            try {
                  ps = sraDatabase.getPacketSet(ingress, egress);
            } catch (PathNotPossibleException e) {
                  throw e;
            }

            if (ps == null) {
                  throw new RouterConfigNotPresentException();
            }

            // mask of null means single IP address - no range
            Range src = ps.convertIPtoIntegerRange(srcip, null);
            Range dst = ps.convertIPtoIntegerRange(dstip, null);

            int sourcePort = Integer.parseInt(srcport);
            int destPort = Integer.parseInt(dstport);
```

```
        int proto = Integer.parseInt(protocol);


        //Conduct the validation test
        for (int i = 0; i < ps.tupleArray.size(); i++) {
              Tuple t = (Tuple) ps.tupleArray.get(i);
              if (src.lower < t.sourceIP.lower
                    || src.upper > t.sourceIP.upper
                    || dst.lower < t.destinationIP.lower
                    || dst.upper > t.destinationIP.upper
                    || sourcePort < t.sourcePort.lower
                    || sourcePort > t.sourcePort.upper
                    || destPort < t.destinationPort.lower
                    || destPort > t.destinationPort.upper
                    || proto < t.protocol.lower
                    || proto > t.protocol.upper) {
                    continue;
              }
              return true;
        }
        return false;
  }

  // opens netflow file and "validates" every line
  public void startValidation() {
        BufferedReader br = null;

        try {
              br = new BufferedReader(new InputStreamReader(new
              FileInputStream(
              netflowFile)));
        } catch (FileNotFoundException e) {
              e.printStackTrace();
              System.exit(-1);
        }

        try {
              String netflowLine = null;
              int errorCounterRCNP = 0;
              int errorCounterPNP = 0;
              int totalErrors = 0;
              int notValidCounter = 0;
              int validCounter = 0;
              int totalCounter = 1;
              int skippedCounter = 0;
              int totalTested = 0;
              while ((netflowLine = br.readLine()) != null) {
                    Object[] fields = parseLine(netflowLine);

                    if (fields == null) {
                          skippedCounter++;
                          continue;
                    }

                    String srcip = (String) fields[0];
                    String srcport = (String) fields[1];
                    String dstip = (String) fields[2];
```

```
                        String dstport = (String) fields[4];
                        String protocol = (String) fields[4];
                        String ingress = (String) fields[5];

                        try {

                              if (fields.length == 8) {
// multiple egresses
      String[] egresses = parseMultipleEgress((String) fields[7]);
      boolean result = validate(srcip, srcport, dstip, dstport,
      protocol, ingress, egresses);

      if (result == false) {
            notValidCounter++;
            System.out.print("Validation failed on line "
            + (totalCounter) + " of netflow input: ");

      System.out.println(netflowLine);
      }
      } else {
// Single egress
      String egress = (String) fields[7];
      boolean result = validate(srcip, srcport, dstip,
      dstport, protocol, ingress, egress);

      if (result == false) {
            notValidCounter++;
            System.out.print("Validation failed on line "
            + (totalCounter) + " of netflow input: ");

      System.out.println(netflowLine);
      } else {
            validCounter++;
            System.out.println("!!!!!!!!!!! Line "
            + totalCounter + " of netflow is valid !!!!!!!!!!");
            }
      }
} catch (RouterConfigNotPresentException rcnpe) {
      System.out.print("Error occured on line " + (totalCounter)
            + " of netflow input - no router config present:");
      System.out.println(netflowLine);
            errorCounterRCNP++;
            } catch (PathNotPossibleException pnpe) {
                        errorCounterPNP++;
                  }

                  totalCounter++;
                  }

      DecimalFormat df = new DecimalFormat("##0.0000");

      totalErrors = (errorCounterRCNP + errorCounterPNP +
      skippedCounter);

      totalTested = (totalCounter - totalErrors);

      double percentValid = ((validCounter / totalTested) * 100);
```

```
double percentNotValid = ((notValidCounter / totalTested) * 100);
double percentTested = ((totalTested / totalCounter) * 100);

double roundedPV = new

Double(df.format(percentValid)).doubleValue();
            double roundedPNV = new
Double(df.format(percentNotValid)).doubleValue();
            double roundedPT = new
Double(df.format(percentTested)).doubleValue();

System.out.println
("================REPORT==========================");
System.out.println("Total passed validation: " + validCounter
                  + " is " + roundedPV + "% of lines tested");
System.out.println("Total failed validation: " + notValidCounter
                  + " is " + roundedPNV + "% of lines tested");
System.out.println("Total lines processed: " + totalCounter);
System.out.println("Total processing errors encountered: "
                  + totalErrors + "\n"
                  + "\t" + errorCounterRCNP
                  + " due to missing router config\n"
                  + "\t" + errorCounterPNP
                  + " due to no path found\n"
                  + "\t" + skippedCounter
                  + " were skipped for missing a field value")
System.out.println("Total lines tested: " + totalTested + " is "
                  + roundedPT + "% of total processed");
System.out.println
("==================================================");
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Burt Lundy (2006). Brief History of Telecommunications: Early Communications and the Telegraph, Computer Science 4556 Business Economics Network Technology Course Notes, Naval Postgraduate School, Spring Quarter 2006.

2. G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson and J Rexford (2005). On static reachability analysis of IP networks, INFOCOM 2005, 24th Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings IEEE, Florida, USA, pp. 2170-2183, vol.

3. Packet Design, Inc. Route Explorer. http://www.packetdesign.com/products/products.htm, last accessed February 2007.

4. E. G. W. W. Wong (2006). Validating Network Security Policies Via Static Analysis of Router ACL Configuration (Thesis, Naval Postgraduate School, December 2006).

5. A.V. Aho, J.E. Hopcroft, and J.D. Ullman (1994). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

6. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest (1990). *Introduction to Algorithms*. MIT Press (McGraw-Hill).

7. Cisco Systems (2007). Cisco IOS NetFlow Data Sheet, http://www.cisco.com/go/netflow, last accessed 23 March 2007.

8. N.G. Duffield, C. Lund, M. Thorup (2005). Learn more, sample less: control of volume and variance in network measurement, IEEE Transactions in Information Theory, pp. 1756-1775, vol. 51, no. 5.

9. N.G. Duffield and C. Lund (2003). Predicting Resource Usage and Estimation Accuracy in an IP Flow Measurement Collection Infrastructure, ACM SIGCOMM Internet Measurement Conference 2003, Miami Beach, Fl, October 27-29, 2003.

10. N. Alon, N.G. Duffield, C. Lund, M. Thorup (2005). Estimating sums of arbitrary selections with few probes, PODS.

11. N.G. Duffield, C. Lund, M. Thorup (2004). Flow Sampling Under Hard Resource Constraints, Sigmetrics.

12. Mascopt (Mascotte Optimization) Project, Tools for Network Optimization Problems. Open source under LGPL. http://www-sop.inria.fr/mascotte/mascopt/index.html, last accessed February 14, 2007.

13. Cisco IP Named ACLs http://www.cisco.com/en/US/products/sw/secursw/ps1018/products_tech_note09186a00800a5b9a.shtml#ipnamacl, last accessed March,2007.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Geoffrey Xie
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

4. Professor Richard Riehle
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California